**Introduction**

**Yocto**

**BitBake**

**Related work + Motivation + Use cases**

**Fuego**

**AGL-JTA**

**LAVA**

**Image Tests**

**LTP**

**Design + Implementation + Testing**

**1. Specification abstraction through tests and meta-data**

Each specification is build around certain systems and requirements and my differ fundamentally from one another even if they try to standardize the same system. Specifications are very dynamic, changing they contents more or less frequently depending on technology advancement and target systems. Ellida Framework tries to find the similarities between Linux specifications using two of the most popular variants - Automotive Grade Linux (AGL) and Carrier Grade Linux (CGL) - and find a suitable abstraction that can be used in a software system.

**Carrier grade Linux - CGL**

CGL is defined sets of requirements and gaps. Each set represents a separate category or objective that aims to provide a different functionality vital to the system and can be made of one or more requirements and gaps. The objectives are:

1. Availability
2. Clustering
3. Serviceability
4. Performance
5. Hardware support
6. Security
7. Standard implementation

A requirement represents an aspect, application or feature essential for providing one of the above functionality that has at least an active or open source implementation, while a gap does not currently have an active or open source implementation.

A requirement contains:

ID: unique identifier composed of some identification elements

Name: short description

Category: one of the objectives

Priority: marks if a requirement has to be implemented or not

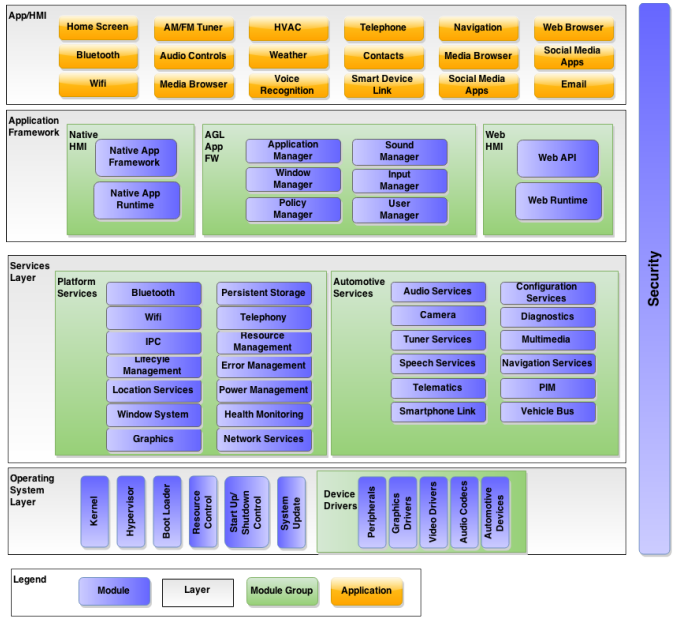
Description: detailed description

Gaps have a similar format.

**Automotive grade Linux - AGL**

Structured as several layers that focus on a specific part of the system. AGL layers are:

1. App/HMI
2. Application framework
3. Services
4. Operating system



Each layer consists of several components and their description, each component might have one or more levels of sub-components.

**TODO: tree representation of AGL**

The conceptual difference between AGL and CGL specifications is that AGL looks in detail at a monolithic system and its main software components - the particular details of a certain kind of system. CGL however has a broader approach referring to its main building blocks as objectives - a dynamic, feature oriented classification of requirements. AGL defines building blocks or modules specifications, CGL defines a set of actions and how they should be behave so that systems main objectives are accomplished.

**2. Framework design and main components**

**Startup system**

The Ellida framework is started via

**Database**

The tests are structured as a directory tree with JSON files representing parts of the specification. Each specification stands as a root directory with the rest of the tree made so that it maps the requirements.

Screenshots of directory trees

**Test Manager**

Role: manage the test database by adding/removing/changing tests

The test manager is the only component with which the user needs to interact (through a GUI), it provides ease of access to the database and formats any new tests so that the framework can use them.

How and where should the mapping between a test file and a .json file be made?

1. Inside the .json specify the path or name of the test
2. Let the manager handle the mapping at each run using the .json ID - how does it know what tests to use?

**[?] Including python modules from other folders seems to be discouraged in Yocto, classes and recipes fully implement the components they need. Is it because so it stays more modular or does it have something to do with the way python imports modules (using sys and append the path to the python path at runtime seems a bad practice)?**

Software used for profiling:

perf

LTTng

systemtap

oprofile

**Evaluation**

<http://swarm.cs.pub.ro/~razvan/wiki/school/diploma>

**Ideas & Further development**

1. Extend testing beyond VMs to physical devices
2. Include Rust as alternative to Python for writing tests
3. Integrate Selenium framework or at least an API that makes usage easy
4. Include ideas from Wind River CGL profiler:
   1. Error Detection and Correction - set of kernel modules that report and handle hardware errors - can be done with systemtap
   2. Kdump - kernel dumps available within the framework to look for possible causes of kernel crashes
   3. Persistent storage - keep data across reboots